Tejas Nitin Joshi
#102181822
tejas.joshi@colorado.edu

ELEC 5623 RTES

LAB 3 REPORT

Understanding and writing a Device Driver

1. Probing the PCI

To probe the PCI bus we use functions from the **pciConfigShow** library which contains routines of PCI bus(IO mapped library)

The routines being used in the programs are as below with the short description:

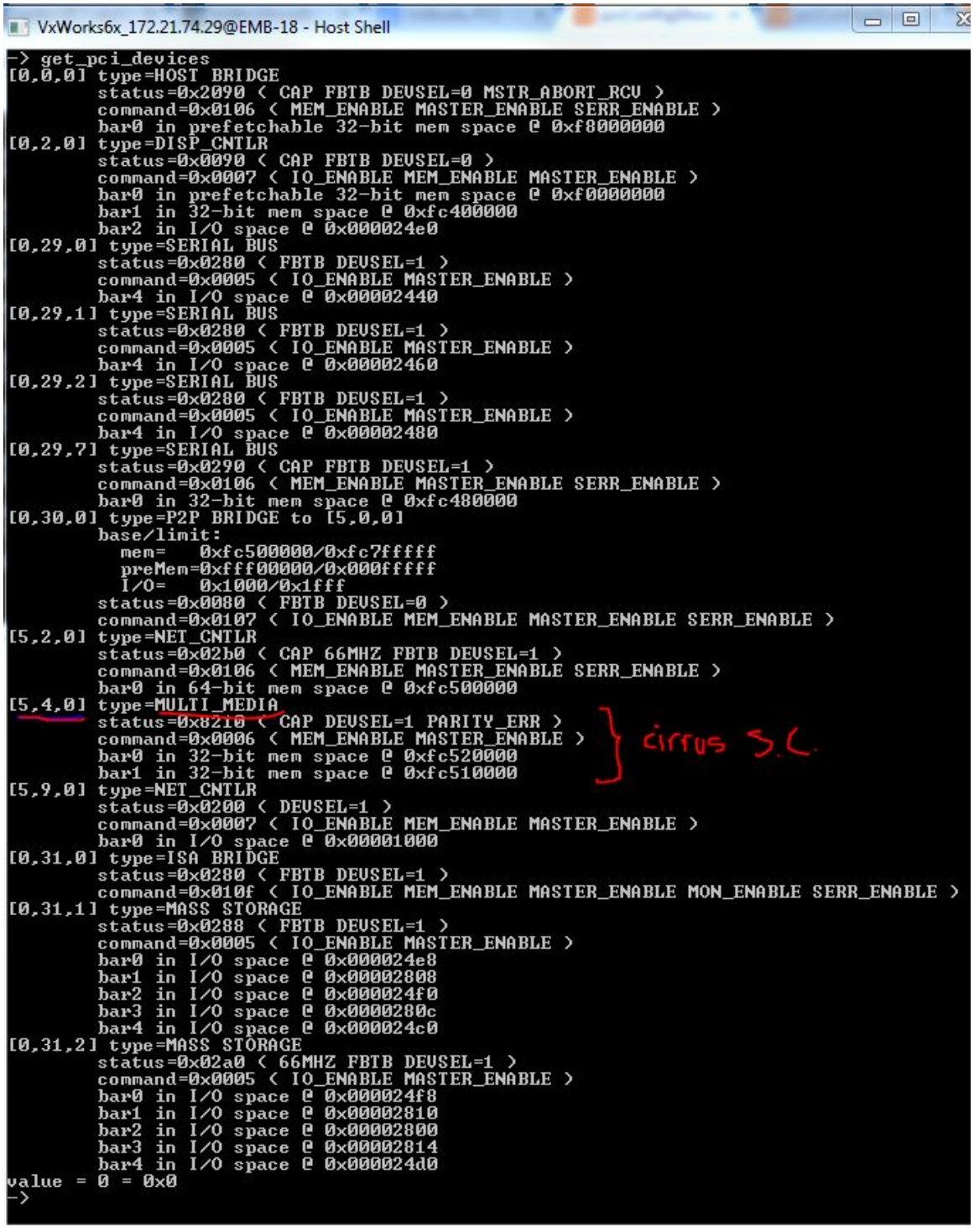
pciDeviceShow( ) - print information about PCI devices

pciFindDeviceShow( ) - find a device by deviceID, then print an information.

pciConfigTopoShow( ) - show PCI topology

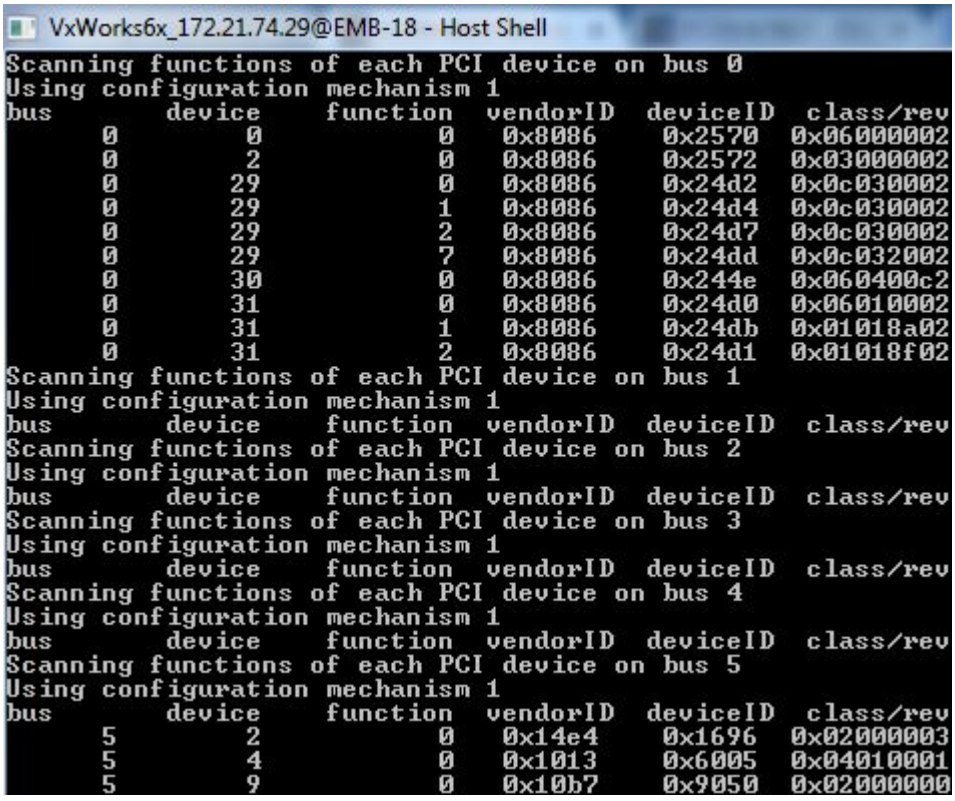
This command traverses the PCI bus and shows what bus contains which device and reports back with the 1)Device Type 2) Command and Status Words 3) PCI to PCI bridges 4)values of Base Address Registers(BARs)

Although this performs a thorough search it does not provide information on Device ID and vendor ID information but provides a more generalized overview of the bus to device connections. The Cirrus logic card here is described as a “Multimedia Device” in the output



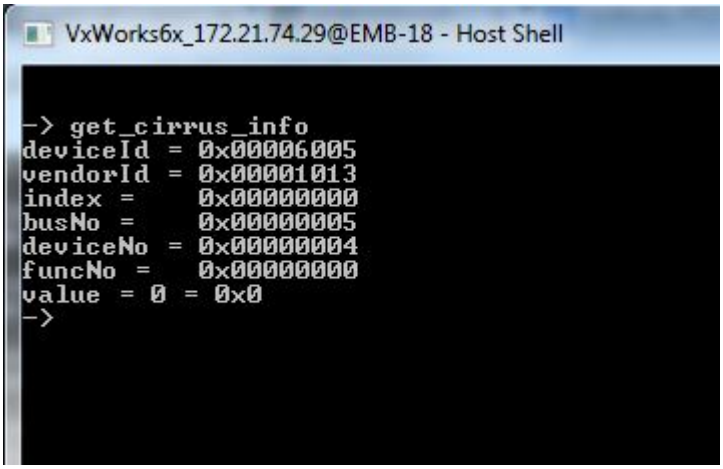
The [540] Here denotes the bus, device and function numbers respectively with BAR info listed below.

For a more comprehensive result all bus numbers are scanned for connected devices using the pciDeviceShow() command which provides much detailed information for each bus and device number and functions of the device as shown below



The function is provided with all Bus Numbers (0-255) to poll for information if existing. The cirrus logic audio card is shown on bus number 5 and can be identified from the device and vendor IDs.

To specifically find a device from the PCI bus if the Device ID and Vendor ID is known (for the soundcard datasheet in our case).



Modifications done to the Driver Code:

All declarations for the variable MAX_LINE specified have be replaced with DAC_BUFFER_SIZE variable in the code

Synchronization Semaphores are used between the Send Task and Recive Task, The Send task is used for the Analog to Digital Conversion.

By Looking at the code I was able to gather that the original driver was meant to record microphone audio and playback the same sound. completing the loop. The code had been stripped down to work for only playback. With missing portions.

The code contains DTC_DMA_Record semaphore which when released by the Sender task would activate the Reciver(Playback) task to perform its operations.

Hence the check for this semaphore had to be removed from the Recv task for proper functioning.

void start_dac(void)

void start_adc(void)

int prog_codec(void)

functions have intlocks initialized but were not released when the routines ended.

The prog_codec initialed the DMA to be run in a Signed format which caused the file data from 0-255 to be clipped at the 128 level.

Hence another change was to write to enable the DMRn_USIGN flag on the BA0_DMR0(Mode Register for DMA Engine 0)

The BA0_DACSR has be configured to 0x04 which sets the playback rate for the Soundcard in this case 0x04 means a playback rate of 11025Hz . Below is a table for other possible values for the playback rates based on the sampling frequency.

DACSR/ ADCSR	Internal Divider	Desired Fs	Actual Fs	Percent Error
5	3072	8000	8000.0	0.000
4	2229	11025	11025.6	0.005
3	1536	16000	16000.0	0.000
2	1114	22050	22061.0	0.050
1	557	44100	44122.1	0.050
0	512	48000	48000.0	0.000

Table 32. Sound System Special Sample Frequencies

I also tested the sound files I had at the 44100Hz which made the files play faster as they were sampled at 11025Hz

The probe function which is responsible for setting up the PCI bus communication and registers also needs to call the hardware initialization code (i.e. make a call to cs4281_hw_init)after the

enabling the bus master and the memory space, in the section denoted by comment “Bringing up Hardware”

The Half Terminal Count Interrupt is by default disabled on the system hence the Interrupt can be enabled by setting the DCRn_HTCIE flag to enable it so Interrupts will fire for the Half Count.

The probe code then initializes the buffer allocations and performs hooking interrupts to vector for DMA operations. After which the DAC initialization function has to be called (missing in the original code) i.e start_dac() function is called .

Execution Flow:

The execution is done using the following steps :

The start function is executed, this probes the bus and sets up PCI bus and registers, the hardware intiailization is done by cs4281_hw_init() function which is called by the probe function and then all the buffers are initialized(vallock’ed) and interrupt hook and DAC initialization is done by setting the specific register.

The correct working code is as attached.

The recvTask still uses the DTC_DMA_Playback flag to ping pong between the two half buffers. And the DTC_DMA_Playback semaphore for synchronization between DMA operations and the loadbuffer operations.

A count flag is set to disable load_buffer function when need for a specific application.

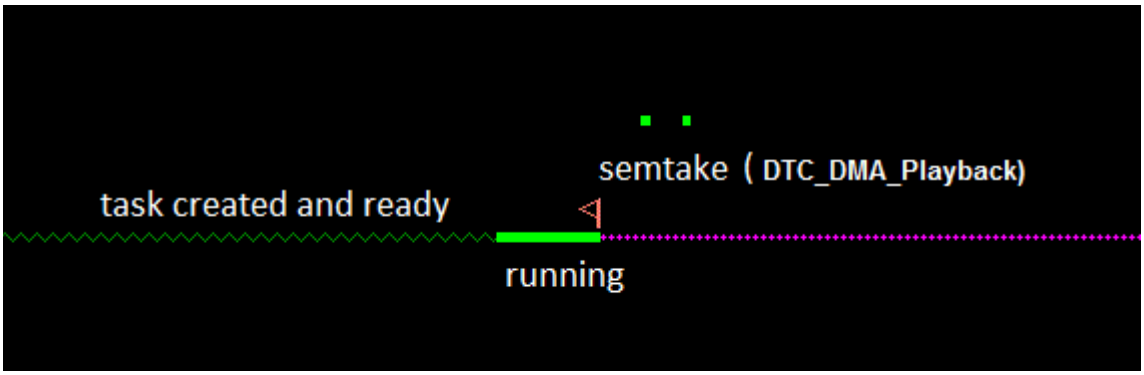
The code for question_2 uses the bugs music array which is a looney tunes theme , I started with the sample data given in pbdata but was very confused regarding the sound I was hearing and what it was supposed to be.

I found that it was easier to convert wav files using audioread() function in Matlab with ‘native’ argument to be more effective in creating the music file arrays. This function requires no mapping of the values generated , it creates the values by default in 0-255 range (mono).

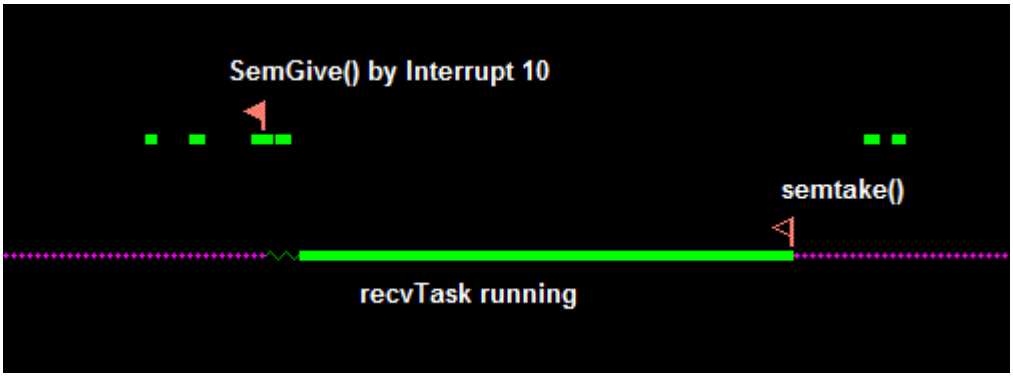
I was able to play the tune(buff) on the soundcard with appropriate quality.

Below is the screenshots from system viewer for which shows the semaphore synchronization between the DMA interrupt and the

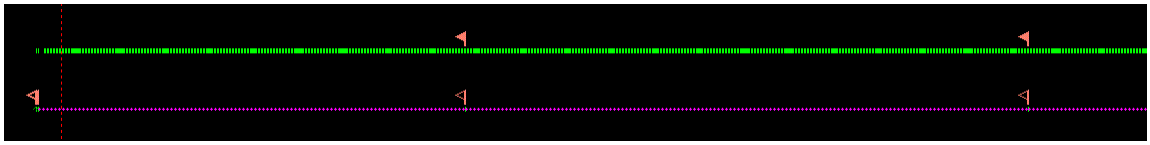
Initial (first run)



Semaphore Syncronization for loadbuffer operation and DMA transfers



Un-zoomed View:



API Implementation:

The API implementation is done by implementing the void sendtobuffer(int *base_addr,int length), which accepts the length and the base address and for the buffer being intput.

A global buffer is maintained named “file_buffer” this buffer is allocated first time when the send tobuffer is called the first time and loads the buffer with the data which is then accessed by the recvTask to load the contents to the DAC_BUFFER. The count flag is used here to prevent simultaneous writing to DAC_BUFFER and file_buffer at the same time, which may cause errors by corrupting data.

The recive task loads the buffer in a while loop hence causing the file to be played in a circular loop. When the user uses the function a second time the function reallocates the “file_buffer” size and copies only the data which is new to the buffer at the end of the previously present data. This causes the buffer to play the combined version of the files in a circular loop.

It is possible to add N number of music file arrays, right upto the point where your memory runs out and gives you a page fault.

Another function named print_buffer_address is implemented which prints out the base address of the music arrays and their sizes which can be used as arguments for the address.

Command to be used when recreating the process:

print_buffer_address

sendtobuffer(&buff, sizeof(buff))

sendtobuffer(&bugs, sizeof(bugs))

sendtobuffer(&swtheme, sizeof(swtheme))

this causes the three files to play in a loop.

Load buffer modification ,

```
void loadbuffer(char *b , int size)
{
  int i;

  for(i=0;i<size;i++)
  {
            b[i] = *(file_buffer+((j+i)% prev_len));
  }j = j +size;

}
```

The load buffer is modified to include the j variable which represent the DAC_BUFFER size and the mod operation with the previous_length monitors the current data size being oaded into the buffer and the mod operation takes care of the incomplete buffer fills.

The Sine Wave generation:

The math behind it :

The playback frequency is 11025Hz hence the maximum frequency that can be reconstructed based on the nyquist principle would be 5512Hz. If the flag is changed for the playback rate to 0x01(i.e 44100Hz) playback, the code for sine frequency still works properly with a higher achievable range of frequencies going upto 22050 Hz.This covers the audible range completely which is from 20Hz to 20KHz.

The number of samples taken from the user requested frequency is decide by

Number of samples = (sampling rate / user requested freq.)

This causes the number of samples to be taken to be variable as samples are only taken to represent one cycle of the sine wave. As we know repeating the one cycle multiple times in a loop will generate a continuous sine wave as the sine patter is symmetrical and repetitive.

The count variable again acts as a mechanism for the buffer completion and instantaneous change initiated by the user to change the sine frequency.

Steps to recreate: